Recognition of Simple Visual Images Using A Sparse Distributed Memory: Some Implementations and Experiments

Louis A. Jaeckel

March 1990

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 90.11

NASA Cooperative Agreement Number NCC 2-408 and NCC 2-387

Research Institute for Advanced Computer Science

Recognition of Simple Visual Images Using A Sparse Distributed Memory: Some Implementations and Experiments

Louis A. Jaeckel

Research Institute for Advanced Computer Science
MS 230-5, NASA Ames Research Center
Moffett Field, CA 94035

RIACS Technical Report 90.11

March 1990

Abstract. In a previous report I described a method of representing a class of simple visual images so that they could be used with a Sparse Distributed Memory (SDM). The images considered consist of several pieces, each of which is a line segment or an arc of a circle. This class includes simple line drawings of alphabetic characters. Each segment or arc is represented by five parameters, and the image as a whole is viewed as an unordered set of segments and arcs. In this report I describe two possible implementations of an SDM, for which these images, suitably encoded, will serve both as addresses to the memory and as data to be stored in the memory. A key feature of both implementations is that a pattern that is represented as an unordered set with a variable number of members can be used as an address to the memory. In the first model, an image is encoded as a 9072-bit string to be used as a read or a write address; the bit string may also be used as data to be stored in the memory. Another representation, in which an image is encoded as a 256-bit string, may be used with either model as data to be stored in the memory, but not as an address. Since an image can be approximately recovered from this encoding, it is possible to do a sequence of iterated read operations, in which the result of each read operation is converted to an image which is then used as the next read address. In the second model, an image is not represented as a vector of fixed length to be used as an address. Instead, I give a rule for determining which memory locations are to be activated in response to an encoded image. This activation rule treats the pieces of an image as an unordered set. With this model, the memory can be simulated, based on a method of computing the approximate result of a read operation. I describe the results of some experiments with a rough small-scale simulation of the second model.

Work reported herein was supported in part by Cooperative Agreements NCC 2-408 and NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

RECOGNITION OF SIMPLE VISUAL IMAGES USING A SPARSE DISTRIBUTED MEMORY: SOME IMPLEMENTATIONS AND EXPERIMENTS

INTRODUCTION

In Jaeckel (1989a) I described a method of representing a class of simple two-dimensional visual images. The images considered are assumed to consist of several *pieces*, each of which is a line segment or an arc of a circle. This class of images is broad enough to include a variety of shapes for line drawings of alphabetic characters. Although I use alphabetic characters as examples in this report, the methods described would apply to any images made of segments and arcs. I also assume that we have a means of identifying the pieces in an image, such as a preprocessor. The number of pieces in an image, NP, is assumed to be not greater than eight. Each piece is represented, or described, by five parameters, in a unique and continuous way, and thus can be thought of as a point in a five-dimensional manifold M. The image as a whole is then represented by an *unordered* set of points in M, one for each piece.

A Sparse Distributed Memory (SDM), proposed by Kanerva (1988), is a memory system that uses addresses that are very long bit strings, or binary vectors, and is able to retrieve stored

data if the retrieval information (the read address) is known only approximately. I assume that the reader is familiar with the basic concept of an SDM. Brief descriptions of it may also be found in Keeler (1988), Kanerva (1989), and Jaeckel (1989b). The memory may be used for various pattern recognition tasks. To do this, there must be a way of encoding the input patterns so that they may be used as read or write addresses. These addresses may be long bit strings, as in Kanerva (1988), or they may have other forms, depending on the problem. To recognize patterns, the SDM is first trained on a set of patterns by writing representations of them to the memory, using an encoding of each pattern as an address to the memory. Then, when it is presented with a pattern which it must try to recognize — that is, identify it or classify it as an instance of one of the stored patterns — the system reads from the memory using an encoding of the pattern as the read address.

This report presents some ways of implementing an SDM so that an image of the kind described above, suitably encoded, can be used as an address to the memory. Since an image is represented by an unordered set of points in M, with different images represented by different numbers of points, a key issue is how to adapt the SDM concept so that the memory can be addressed by patterns represented in this form.

In the first SDM implementation below, called *Model 1*, the unordered set of points in M representing an image is converted to a string of 9072 bits, in a way that is independent of the order in which the pieces may have been listed. Each of these

bits is associated with a *lattice point* in M; a bit is set to 1 if a point representing a piece of the image is near the corresponding lattice point. While these bit strings are very long, this method allows us to use a binary vector of fixed length as an address, as in Kanerva (1988).

I then give a different way of encoding an image as a 256-bit string, to be used as data to be stored in the memory but not as an address. This encoding method could be used with either of the models described below. Since this method involves putting the pieces of an image in an arbitrary order, I will not use it to convert an image to an address. With this encoding method, an image can be approximately recovered from its 256-bit representation; therefore, the memory can be thought of as resembling an autoassociative memory, that is, a memory in which the data word to be written to the memory is the same as the write address. We can then do a sequence of iterated read operations, in which the result of each read operation is converted to an image which is then used as the next read address, as in Kanerva (1988), p. 68. In some cases, this process will converge to a fixed point, resulting in a more accurate response than can be obtained with a single read operation.

In *Model 2*, the second SDM implementation, a different method of addressing is used. In an SDM, any read or write address causes a subset of the memory locations to be activated. In this model, an image is not represented as a vector of fixed length to be used as an address. Instead, I give a rule for

determining the subset of locations to be activated in response to an encoded image. A memory location is defined by choosing three *defining points* at random in M^+, an expanded version of the manifold M. A memory location is activated by an image used as an address if, for each of the three points defining the location, there is at least one piece of the image, represented by a point in M, that is within some given distance of that defining point. This activation rule treats the pieces of an image as an unordered set. With this definition of memory locations, it is possible to compute the approximate number of memory locations activated by both of two images. The memory can be simulated by using this computation to approximate the result of a read operation. I describe a rough small-scale simulation of Model 2 based on this computation.

I then give the results of some experiments done with the simulation. The system was trained with a set of 20 characters, shown in Figure 1. (I will usually use the term *character* to refer to the images that will be stored in the memory.) Each member of the training set was assumed to be written to the memory by adding its 256-bit encoding to the contents of the counters for the memory locations activated by it. Then, since there was some error in the memory's response, due to interference caused by similar characters in the training set, the memory was given a small amount of "retraining", in order to reduce the errors in the response for two of the characters. The system was then able to recognize all of the characters on which it was trained, with only a small amount of interference due to

similar stored characters. It was also able to recognize some images of characters similar to those in the training set (Figure 2). Some other images were too dissimilar to be recognized with one read operation (Figure 3). Some experiments were also done with sequences of iterated read operations, to see whether the system would converge to a fixed point representing a stored character. In some cases, the memory was able to recognize a character after one or two iterations, or to improve the accuracy of its initial response. In other cases, the memory was not able to recognize the character even after iterating.

Further experiments could be done to find out how many characters can be stored in the memory before it becomes overloaded, or how similar the characters can be to one another and still be distinguishable. Experiments could also be done to test the performance of the system with various character sets, design parameters, and methods of retraining the system to improve its performance. I will indicate some possible design alternatives and directions for further work.

In order to give some perspective, I will at times discuss various ways to accomplish a particular task. But in experimenting with and simulating the system, I usually tried to use a design option that is relatively simple mathematically and conceptually, so that it will be easier to understand and to give rough mathematical estimates of its performance. Various refinements and more complex options would probably give better performance, especially with a large character set, but they are not as amenable to simple mathematical estimates of performance.

Different methods will have to be experimented with to see what works best under various conditions.

SDM IMPLEMENTATION, MODEL 1: ENCODING AN IMAGE AS AN ADDRESS

I will now describe a method for implementing an SDM system based on the ideas in Jaeckel (1989a) for representing and encoding the pieces of an image. Another method will be described in a later section. The underlying principle in both methods is that an image consists of an *unordered* set of several pieces, each of which is a line segment or an arc of a circle, and is represented by a point in the manifold M. As explained in Jaeckel (1989a), p. 40, there is not a natural, continuous way to order the pieces. A segment or an arc is described by five parameters: the X and Y coordinates of a center point for the piece, relative to the other pieces; the relative size of the piece; and an unordered pair of angles (or points on a circle) which jointly represent the orientation and shape of the piece. See Jaeckel (1989a), p. 18-25. I assume that we have a means of finding the pieces in an image; the problem of building a preprocessor to find the pieces is discussed in Jaeckel (1989a), p. 42-48.

To implement an SDM, we must be able to use an image as an address to the memory, both for writing and for reading, and also as data to be stored in the memory. Note that it is not necessary to use the same method of encoding the images both for addressing and for data storage. Encoding the images for use as

data will be discussed in the next two sections. To use an image as an address, I will encode it as a bit string, or binary vector, using a method that is based on the representation of an image as a set of points in M. With this method, the length of the bit string will be the same for all images, regardless of the number of pieces the image has; a small change in the parameters describing the pieces will produce only a small change in the bit string; and the pieces will be treated as an unordered set. (A similar problem of encoding an unordered set of variable size as a bit string of fixed length is dealt with in Kahan et al. (1987), p. 276, in their work on character recognition.)

I begin by choosing a large set of points spanning M, that is, a set of points spread uniformly throughout M so that no point in M is very far from one of these selected points. As an example, I will use the set of 9072 *lattice points* described below. An image will be encoded as a string of 9072 bits, that is, a binary vector, in which each bit position, or coordinate, corresponds to one of the lattice points. For a distance function in M, I will use the Euclidean (L2) metric, as defined in Jaeckel (1989a), p. 30-32. I then choose a value R to be used as the radius of a sphere in M (the set of all points in M within R of some point).

Given a set of points in M representing the pieces of an image, consider the set of spheres in M of radius R, whose centers are the points representing the pieces of the image. Note that these spheres may overlap, or extend beyond the borders of M. For each lattice point lying in one or more of these

spheres (that is, in the union of the spheres), assign a 1 to the corresponding bit. All bits corresponding to lattice points not lying in any of the spheres are assigned a 0. We now have a 9072-bit representation of the image: *A bit is 1 if and only if its corresponding lattice point is within R of one or more of the points representing pieces.* Note that the order in which the pieces may have been listed makes no difference, and that the length of the bit string is the same, regardless of the number of pieces in the image.

If R is such that a sphere of radius R about a point in M representing a piece of an image contains a few hundred lattice points, then if the piece is altered slightly, causing a small movement in the point in M representing it, the sphere about the moved point will contain most of the lattice points that were in the original sphere, and a few that were not. Thus there will be only a small change in the bit string representing the image. In other words, this method of representation is *continuous* in the sense defined in Jaeckel (1989a), p. 14. Note that even a very small movement of a point in M will probably cause some differences as to which lattice points are in the sphere about the point.

There is a kind of duality here: Instead of spheres about the points representing the pieces of an image, we can consider spheres of radius R whose centers are the lattice points. For any of the 9072 bits, the bit is assigned a 1 if one or more of the points representing the pieces of an image lie in the sphere of radius R about the corresponding lattice point. Thus each bit

is analogous to a "feature detector", whose receptive field is the set of all segments and arcs that are represented by points within R of the lattice point for that bit. (These spheres are like the intervals for which a particular bit is set to 1, defined in Jaeckel (1989a), p. 36.) Note that these feature detectors have overlapping receptive fields. This method of encoding is somewhat like a concept called "coarse-coding" by Hinton (1981), p. 1094.

If R is a fixed number, then the number of 1's in the bit string for an image will be roughly proportional to the number of pieces in the image. If, however, we want to keep the number of 1's more or less constant (an option that may be important in some SDM designs), we can make R a function of the number of pieces. Something like this will be done in Model 2 below.

I will choose a set of 9072 lattice points in M as follows: When the pieces of an image are each represented by five parameters as in Jaeckel (1989a), p. 18, the values for the first three parameters for each piece (the X and Y coordinates of the center point of the arc, and the size of the arc) all lie between 0 and 1. I choose six numbers in each of these three unit intervals, for example, 0, 0.2, 0.4, 0.6, 0.8, and 1. In the Cartesian product of these three intervals, that is, the set of all possible combinations of values for the first three parameters, the combinations of the chosen values above define $6^3 = 216$ points.

Now consider the Möbius strip of all possible unordered pairs of angles (or pairs of points on a circle) representing the

orientation and shape of an arc (Jaeckel, 1989a, p. 26). I will choose 42 lattice points in this set. Imagine 12 points, numbered 1 through 12, arranged counterclockwise around a circle, beginning at "3 o'clock". If we choose unordered pairs of points from among these 12 points such that the two points are at least 90^{0} apart (as in Jaeckel, 1989a, p. 24), we see that there are 42 possible unordered pairs, beginning with {1,4}, {1,5}, . . . {1,10}; then {2,5}, . . . {2,11}; and so on, up to {9,12}. (Each point may be paired with seven others, giving $12 \times 7 = 84$ ordered pairs; this number must be divided by two since each unordered pair corresponds to two ordered pairs.) Each of these 42 pairs corresponds to a point on the Möbius strip. These points are evenly spread out on the strip. Because of the way in which the unordered pairs represent orientation and shape, these 42 points are arranged on the strip in a sort of diagonal pattern: Moving from one of these points to one of its nearest neighbors corresponds to changing one of the angles describing an arc by 30^{0} while holding the other angle fixed; this is a diagonal motion on the strip, changing both the orientation (longitude) and shape (latitude) of the arc. (See Figure 2 in Jaeckel, 1989a.)

In Jaeckel (1989a) I defined M to be a subset of the Cartesian product of three line segments and a Möbius strip. For simplicity, I will now change the definition of M to be the entire five-dimensional Cartesian product, although, as explained in Jaeckel (1989a), p. 25-26, some of the points in this set do not correspond to possible arcs.

Since M is now the Cartesian product of the three unit intervals and the Möbius strip, I can define a set of lattice points in M by taking each possible combination of the six values for the first three parameters, combined with each of the 42 points on the strip, giving a total of $216 \times 42 = 9072$ lattice points in M. Every point in M is near one of these lattice points. If we use the distance function defined for Model 2 below, then each lattice point is one unit of distance from its nearest neighbors in every direction.

There are a number of possible variations on the above scheme. Since discussing alternatives may help give some perspective, and since I may want to experiment with some of these alternatives, I will mention a few of them here.

The number and spacing of the lattice points gives us a certain degree of resolution, which might be measured by comparing the difference in the encoding of two points in M that are very close to each other. This difference depends on which lattice points are in the sphere of radius R about one point, compared to the other. If we want higher resolution, we could have more lattice points, which would mean longer bit strings; or we could give up some resolution in order to have shorter bit strings. If we redefine the metric to give greater weight to some of the parameters, as was suggested in Jaeckel (1989a), p. 30, we could choose lattice points so that their spacing would be consistent with that metric.

Another way to choose lattice points on the Möbius strip is to choose 40 points arranged in more of a square pattern, as

follows: Choose eight points on the "equator" of the strip (the set of points representing line segments with different orientations) and then, for each of those points, choose four other points, two above and two below the point on the equator, representing arcs with the same orientation but with different shapes. The result will be eight sets of five points, where each set of five points lies on a line perpendicular to the equator. The 40 lattice points can be chosen as follows: Imagine 16 points, numbered 1 through 16, arranged counterclockwise around a circle. Choose unordered pairs of these 16 points, such that the angle between the points in a pair is either 90^{o}, 135^{o}, or 180^{o}. Since each point may be paired with five others, there are 80 such ordered pairs, and therefore 40 unordered pairs, corresponding to 40 points on the strip arranged as described above. If we combine each of these 40 points with each of the 216 possible combinations for the other parameters, we will have 8640 lattice points in M. A possible reason for using this set of lattice points is that it allows us to think about segments and arcs in a way that separates their orientation from their shape. (See Jaeckel, 1989a, p. 28.)

A problem with these sets of lattice points is that since they are the vertices of five-dimensional cubes in M, the center of each cube is relatively far away from any lattice point. While there is probably an optimal way to choose a set of points in a five-dimensional space so that no point in the space is far from a lattice point, the advantage of the lattice points defined above is that they are easy to work with.

DEFINING MEMORY LOCATIONS FOR MODEL 1

To define an SDM, we must define a set of potential memory locations, a random sample of which will be implemented and called *hard locations*, as in Kanerva (1988). In Kanerva's basic design, each point in the address space represents a potential memory location, and the set of read and write addresses for which the memory location represented by the point x would be activated is the set of address vectors that are within some given Hamming distance of x. The address space for Model 1 is a 9072-dimensional binary vector space. However, since an image is encoded as a bit string that somewhat resembles a union of five-dimensional spheres, the set of possible addresses is only a small part of the entire 9072-dimensional address space. So, instead of choosing the addresses for the hard locations at random throughout the address space, it might be better to define hard locations using only addresses that are close to the set of possible addresses for images, so that we do not have a large number of hard locations that are far away from any possible address. Keeler (1988), p. 321-24, has suggested choosing the addresses of the hard locations so that their distribution is like that of the addresses corresponding to the images that will actually be encountered. If this is done, the system should use the memory locations more efficiently. The activation radius would then have to be adjusted so that when reading or writing, a desired number of hard locations would be activated. One way to choose such addresses for hard locations would be to create

artificial images made up of several pieces chosen at random, encode them as address vectors, and use these vectors as addresses of hard locations. This is somewhat analogous to the method of defining memory locations in Model 2 below.

In two recent technical reports (Jaeckel, 1989b, 1989c) I described some alternative designs for an SDM. Any of those designs could be used here. Because of the very long addresses used in Model 1, the address decoding in some of those designs, such as the "selected-coordinate design" and the "hyperplane design", would be simpler than in Kanerva's design.

The next issue in designing an SDM is what to store in the counters for the memory locations activated during a write operation. This depends on how we are going to read from the memory. If we intend to use the memory as an autoassociative memory — that is, when we write to the memory the data word stored is the write address — we would need 9072 counters for each hard location. We could then do a sequence of iterated read operations to attempt to converge to a fixed point, as in Kanerva (1988), p. 68. With this number of counters, we could also store sequences by writing at each address the next address in the sequence (Kanerva, 1988, p. 80). However, such a large number of counters would require a lot of hardware. An alternative method of storing images, requiring only 256 counters per hard location, will be described in the next section.

If we are doing supervised learning, that is, if we know the response to be given for each member of the training set, then when we write to the memory we can store some kind of identifier

or code (such as an ASCII code) for each character to be stored in the memory, rather than the entire write address. These identifiers would require only a small number of bits, compared to 9072 (or to 256), and the memory system would need correspondingly less hardware. The identifiers could have some redundancy built into them, at the cost of a few extra bits, so that the memory would only have to recover most of the bits correctly, instead of every bit. If we intend to do a single read operation to attempt to recognize an image of a character, all we need to store are these character identifiers. On the other hand, if we want to do iterated read operations to improve the memory's response, we could store both the write addresses and the identifiers. In either case, reading from the memory gives us an identifier in response to a read image; we can then use a look-up table or some other means to find the relevant information associated with that identifier.

If we store 9072-bit address vectors to be used in iterated read operations, then when we read from the memory we obtain 9072 sums, as a result of adding the contents of the corresponding counters for the activated hard locations. What do we do with these sums? That is, how do we interpret the result of the read operation? We could compare each sum to a threshold, for example an overall average of the bits that have been written to that bit position, and then convert the sums to 1's or 0's, or we could scale them in some other way. If we are using the memory as an autoassociative memory, and we read at an address near the address of a stored character, we expect to get an approximation

to the write address of that character; that is, we expect to find large sums for the bit positions corresponding to lattice points close to the points in M representing the pieces of the stored character. So, when we read, we want to find sets of bit positions corresponding to clusters of lattice points in M, all of whose sums are large. In other words, we want to find roughly spherical hills or plateaus in a discretized five-dimensional space. And when we find such a plateau, we need to estimate its center and its extent, and maybe its mass. Finding these plateaus could be a difficult pattern recognition problem, or search problem, in itself.

There are a number of possible approaches. We could look for relative maxima among the sums, or we could first smooth the sums by computing a linear combination of each sum with its nearest neighbors, and then look for relative maxima. These operations might be done quickly by parallel computations. Once we find a potential plateau, some amount of computation will be required to estimate its parameters, and to decide whether it really represents a piece of a stored character. When searching for plateaus, we might use the pieces of the image being read as starting points. If the read image is similar to a stored character, the pieces of the read image should be near the corresponding pieces of the stored character, so we might be able to move toward them by an iterative procedure. We would also need to check to see whether or not there is a piece of the stored character near each piece of the read image, and whether there are apparent pieces in the memory's response that do not

correspond to any of the pieces of the read image.

Actually, when we are doing a sequence of iterated read operations, we may not need to interpret the sums at each step as pieces of an image. All we need at each intermediate step is the next read address to try, and that does not have to correspond exactly to an address of a possible image; we only need a set of values for the address bits that is likely to be closer to the address to which we are trying to converge. At the end of the process we may want to use the sums to identify the pieces of the character we have found, so that we can reconstruct it directly. Or, if an identifier was stored with each stored character, then the identifier found by a sequence of read operations can be treated as the memory's response to the read image.

The main disadvantage of this model is the large number of bits in the addresses. This is a consequence of trying to use a vector of fixed length to grasp all of the pieces of an image at once, without imposing an ordering on them. Moreover, if we want to improve the resolution of the system, that is, its ability to distinguish between points close to each other in M, we would need a set of lattice points forming a finer mesh; since M is a five-dimensional manifold, this would require a great increase in the number of lattice points, and hence in the length of the addresses. However, I believe that with the numbers used above, the system would have adequate resolution for recognizing simple images. The capacity of the system to store a large number of characters might well be constrained more by the number of hard locations implemented, than by the number and spacing of the

lattice points.

A small-scale prototype SDM has been constructed at Stanford University (Flynn et al., 1988). It allows for addresses of up to 256 bits. While this is a long way from 9072, it might be possible to implement a scaled-down version of Model 1 above for performing simple demonstrations and experiments, by choosing 256 lattice points in M and assigning an address bit to each. Although such a system could be expected to have poor resolution, it would be interesting to know what could be achieved with such limited resources. One possibility would be to use images made only of line segments, instead of segments and arcs, in which case the pieces would be represented by points in a four-dimensional manifold.

A REPRESENTATION OF AN IMAGE TO BE USED ONLY AS DATA

In Jaeckel (1989a) I showed that if we could assign an order to the pieces of an image, we could represent it by a bit string containing only a few hundred bits, rather than several thousand. If we use 30 bits per piece and allow up to eight pieces, as discussed in Jaeckel (1989a), p. 39, we can represent an image by a string of up to 240 bits. Note that an image can be reconstructed, at least approximately, from such a representation. In either of the SDM implementations in this report, we could use a representation of this kind for the data to be stored in the counters when writing to the memory. Since this representation puts an ordering on the pieces of an image, I

will not use it to convert an image to an address. This method is used in the simulation of Model 2 described below. Since we can approximately reconstruct an image from this form of representation, the model can act like an autoassociative memory, and in some cases we will be able to do a sequence of iterated read operations to attempt to converge to a fixed point, as in Kanerva (1988), p. 68. Compared to the autoassociative version of Model 1 above, it will be much easier with this representation to interpret the result of a read operation, and fewer counters will be needed for each memory location.

The representation of an image of a character that I will use as a data word to store in the counters for the memory locations is a 256-bit binary vector constructed as follows: The first 240 bits are divided into eight 30-bit *blocks*, or *piece-positions*, each of which can contain one piece, encoded as a 30-bit string. Some of the blocks may be blank. Each 30-bit block contains sub-blocks of six bits each for the X and Y coordinates of the center of the piece, six bits for the size, and 12 bits for the pair of angles representing orientation and shape. These parameters are encoded as bit strings as described in Jaeckel (1989a), p. 34-39. When an image of a character is to be written to the memory, it is assigned one of the eight blocks at random as a starting block. A piece is placed in that block and in each of the succeeding blocks, wrapping around from the eighth block to the first block if necessary. The order of the pieces does not matter, except in the situation described below. Unused blocks are filled with 0's. A random starting block is

used so that when many characters are stored, each of the eight blocks will be used to store about the same amount of data. The remaining 16 bits are used to indicate which blocks have been filled with pieces of the character. In the first eight of these bits, the bit corresponding to the starting block is assigned a 1, and in the last eight bits, the bit corresponding to the final block filled is assigned a 1. The other 14 bits are set to 0.

A write operation consists of determining which memory locations are activated by the write address, and then adding the components of this 256-bit string to the numbers already in the data counters for the activated locations. Thus there must be 256 counters for each memory location. If we use more bits to represent an image, which would require more counters, we could increase the resolution of the system by using more bits per block, or we could increase the capacity by having more blocks, so that any two characters would be less likely to have to share the same blocks.

Although I assign an ordering to the pieces in an image when I store this representation of it in the counters for the activated memory locations, I will continue to treat an image as an unordered set of pieces when I use it as an address to the memory. In both Model 1 and Model 2 the activation rule for the memory locations is such that the set of memory locations activated by an image does not depend on the order in which its pieces may have been listed. The reason for this is to avoid the problem of discontinuities caused by assigning an ordering to the pieces in an image. (See Jaeckel, 1989a, p. 40.)

When I assign an ordering to the pieces of a character for storing in the counters, it does not matter how the pieces are ordered, except in the following situation: If two or more similar instances of a character are included in the training set and written to the memory, such as several instances of upper case "A", each consisting of three line segments in the usual way, the corresponding pieces must be encoded in the same order and in the same piece-positions, so that when we read from the memory the stored data words will reinforce each other. If this were not done, differently ordered instances of the character would partially cancel each other out when stored in the memory. Therefore, this method requires that we know which items in the training set are to be considered as different instances of the same character. The rule above does not apply, however, to dissimilar versions of a character, such as the two distinct ways to make a lower case "g"; these should be viewed as two different characters that happen to have the same name.

SDM IMPLEMENTATION, MODEL 2

I will now describe a simple design for an SDM, for which unordered sets can be used as addresses. Instead of representing an image as a vector to be used as an address, I will give an activation rule for determining which memory locations are to be activated by an image used as an address. I will then describe a rough approximate simulation that was done to test the performance of the memory, and give some examples.

We will need a distance function to measure the distance between points in M. I will use the Euclidean (L2) distance, computed as described in Jaeckel (1989a), p. 31. The distance function is adjusted so that for each of the first three parameters, 0.2 is considered as one unit, and for each member of the pair of angles representing orientation and shape, 30^{o} is one unit. These choices reflect a judgment about the relative importance of a change in one parameter compared to a change in another parameter; see Jaeckel (1989a), p. 30. (Because of the representation of orientation and shape as a pair of angles, changing one angle by 30^{o} would correspond to moving one unit of distance diagonally on the Möbius strip. This is not the same as changing both angles by 15^{o}; that would correspond to moving a distance of 0.7071.)

Now I will make another change in the definition of the manifold M. I will expand M beyond its borders somewhat, by allowing each of the first three parameters (X, Y, and size) to lie in the interval [-0.2, 1.2] instead of [0,1], and by adding a strip to the edge of the Möbius strip component of M so that on the edge of the expanded strip a point corresponds to a pair of angles whose difference is 47.57^{o} instead of 90^{o}. The effect of these changes is to extend the borders of M in all possible directions by one unit of distance, according to the measure of distance defined above. I will call this expanded set M^{+}. The reason for doing this is that when we look at a sphere about a point in M representing a piece of an image, if the point is near the boundary of M a substantial part of the sphere about

it might extend beyond the original borders of M. This would reduce the effective volume of the sphere. Because of the way in which the memory locations are defined in this model, the number of memory locations activated by an image depends on the volumes of spheres about the points in M representing its pieces. With M^+, the expanded M, the volume of a sphere about a point near the boundary of M will not be reduced by quite as much.

I define a memory location and its activation rule as follows: Choose three points at random in M^+. These will be called the *defining points* of the memory location. Note that these points may be any points in M^+, not just lattice points. (However, choosing them from among a set of lattice points might simplify some of the computations.) A write or a read address consists of an image represented by an unordered set of pieces, each of which is represented by five numerical parameters. These parameters will be used as numbers in the addressing process; they will be converted to bit strings only for the purpose of storing them as data. A radius R is chosen as explained below, depending on the number of pieces in the image. *A memory location is activated by an image used as an address if each of its three defining points is within R of at least one of the points in M representing the pieces of the image.* Note that more than one defining point may be within R of the same piece of the image, but each defining point must be within R of some piece. (An alternative design, discussed below, would require each defining point to be within R of a distinct piece of the image.)

We can think of each memory location as having an address

decoder that computes the distance between each defining point and each of the NP pieces of the image used as the address, creating a 3×NP matrix. If each of the three rows of the matrix contains an entry less than or equal to R, the location is activated. It does not matter if some of those entries are in the same column. (For the alternative mentioned above, those three small entries would have to be in distinct columns as well as in distinct rows.) This model is very similar to a version of Model 1 above: If in Model 1 we make R a function of NP and define memory locations as in the "hyperplane design" described in Jaeckel (1989c), p. 17, then the rule for activating a memory location is essentially the same as in Model 2.

I assume that a large number of memory locations, say 100,000, are chosen at random and implemented as hard memory locations. I will use a uniform probability distribution for the points in M^+ defining the memory locations; this may not be the best distribution to use, but it will enable us to do some simple computations. The radius R will be chosen to be a function of NP so that the proportion of memory locations activated by an image is approximately a predetermined amount, say 1/1000, as in some examples given by Kanerva (1988). Consequently, about 100 memory locations would be activated by an image. The actual number will vary, because of the random choice of the memory locations. With these numbers in mind, I decided to use three defining points for each memory location. Using more than three would require larger values for R, resulting in poorer resolution, and using fewer might make it more difficult to distinguish between images that

have some pieces in common.

If the radius used is very large, then it will be difficult for the system to distinguish between images. On the other hand, for a given number of memory locations, the radius must be large enough so that a substantial number of them are activated by an image. Expanding M to M^+ results in using a larger radius than would have been used with M, but was necessary so that roughly the same number of memory locations would be activated by any image. The effect of using a larger radius is that the estimates of the memory's performance will err on the conservative side, since a larger radius means poorer resolution.

For a given image, let U be the union (in M^+) of the spheres of radius R about the points in M representing the pieces of the image, that is, the set of all points within R of at least one piece. Let p be the volume of U divided by the volume of M^+; this is the probability that a point chosen at random in M^+ will lie in U. So if three points are chosen at random to be the defining points of a memory location, the probability that all three lie in U is p^3. But this is the condition for activating a memory location — that each of its defining points be within R of some piece of the image. (Note that the three points defining a memory location do not have to lie in different spheres.) Therefore, the probability that a memory location selected at random is activated by an image is p^3, and the expected number of memory locations activated is p^3 times the total number of hard memory locations. For example, if $p = 0.1$, then approximately 1/1000 of the memory locations would be activated

by an image.

Instead of looking at spheres of radius R about the pieces of an image, we can look at the spheres of radius R about the three defining points of a memory location. This is analogous to the duality mentioned earlier. In this view, a memory location is activated by an image if each of those three spheres contains a piece of the image.

To simplify the computations, I will assume that U is approximately a union of disjoint spheres in M^+. For moderate values of R, and for the images of characters used in the experiments, the overlaps between the spheres of radius R about the points representing the pieces of an image are fairly small, since the points are not very close to one another. Also, most of the volume of these spheres lies within M^+; this was the reason for expanding M. Therefore, the volume of U is approximately NP times the volume of a five-dimensional sphere of radius R.

I will show that the volume of M^+, computed with the measure of distance defined above, is 18168.90. This is considerably more than the volume of M, which can be shown to be 4500; expanding M in four of its five dimensions greatly increases its volume. Although the Möbius strip curves back on itself, locally it is like a plane surface. Consequently, M^+ is locally like five-dimensional Euclidean space. Since the range of each of the first three parameters in M^+ is from -0.2 to 1.2, corresponding to a distance of 7, the volume of the Cartesian product of these three intervals is $7^3 = 343$. This volume must be multiplied by

the area of the expanded Möbius strip. Since the strip is just a rectangle with a twist, we need to find its length and width. Suppose we travel once around the "equator" of the strip (the points representing line segments); this corresponds to a 180^{o} rotation of a line segment. If we change the orientation of a line segment by 30^{o}, we change both of the angles that represent its position on the strip by 30^{o}. Since each angle is displaced by one unit of distance, and the distance function is defined so that these two angle parameters are orthogonal to each other, changing both of them by 30^{o} corresponds to a motion of $\sqrt{2}$. Therefore, moving all the way around the equator corresponds to a motion of $6\sqrt{2}$. Now if we move from the equator, perpendicularly to it, to the edge of the strip as originally defined (corresponding to bending a line segment into an arc), we move the two angles representing a point on the strip toward each other, shrinking their difference from 180^{o} to 90^{o}. In other words, each angle is changed by 45^{o}, or 1.5 units of distance. Therefore, the distance from the equator to the edge is $1.5\sqrt{2}$, and so the width of the strip is $3\sqrt{2}$. Since I expanded the strip by adding one more unit of distance to the edge, the expanded width is $3\sqrt{2} + 2$. (Since moving one unit of distance outward from the equator can be shown to correspond to changing each angle by $15\sqrt{2}$ degrees, a point on the edge of the expanded strip corresponds to a pair of angles whose difference is 47.57^{o}.) Therefore, the area of the expanded strip is $6\sqrt{2}\cdot(3\sqrt{2} + 2)$, and so the volume of M^{+} is 18168.90.

If we want 1/1000 of the memory locations to be activated

by an image, the radii of the spheres about the points in M representing the pieces of the image must be such that the volume of their union will be approximately one tenth of the volume of M^+, or 1816.89. I will first compute the radius of a single sphere with this volume. The volume of a five-dimensional sphere of unit radius is 5.263789. (See the Appendix.) Since the volume of a sphere is proportional to the fifth power of the radius, we find that the radius must be 3.218148. If an image has NP pieces, and if the NP spheres about the points representing them are assumed to be disjoint and to lie within M^+, then the volume of each sphere must be 1816.89/NP. To find the radius of a sphere with this volume, I divide 3.218148 by the fifth root of NP. The program described below uses radii computed in this way.

The *access overlap* for two images is the set of memory locations activated by both. When we read from an SDM, we compute a vector of sums by adding the contents of the corresponding counters for the activated memory locations. Therefore, when we write to the memory at one address, and then read from the memory at another address, the size of the access overlap for the two addresses determines the number of copies of the written data word that are included in the sums computed in the read operation. It follows that the vector of sums computed in a read operation is a weighted sum of the data words stored in the memory, in which the weights are the sizes of the access overlaps for the write addresses and the read address (assuming that none of the counters had reached its ceiling during writing

to the memory). See Kanerva (1988), p. 67. If two images are near each other, that is, if the points in M representing one image are near the corresponding points in M for the other image, then their access overlap will be large. On the other hand, two very different images will have a small access overlap. Consequently, if the read address is near one of the addresses at which data was stored in the memory, and if no other write address is near the read address, then the result of the read operation will be approximately proportional to the data stored at the nearby write address, mixed with some "random noise".

If U_1 and U_2 are the respective unions of spheres for two given images, as described above, then a memory location is activated by both images if and only if all three of its defining points lie in the intersection of U_1 and U_2. The expected number of such memory locations is therefore proportional to the cube of the volume of this intersection. More precisely, if q is the volume of the intersection divided by the volume of M^+ — the probability that a randomly chosen point lies in the intersection — then the probability that three points chosen at random will lie in the intersection is q^3; hence the expected number of memory locations in the access overlap is q^3 times the total number of hard memory locations. Since U_1 and U_2 are each unions of spheres, their intersection is the union of the pairwise intersections of each sphere in U_1 with each sphere in U_2. Based on the assumptions about these spheres made above, we may assume that these pairwise intersections are mostly disjoint from one another and that they lie mostly within M^+; therefore, the volume

of the intersection of U_1 and U_2 may be roughly approximated by the sum of the volumes of the individual pairwise intersections of the spheres. A formula for the volume of the intersection of two five-dimensional spheres is derived in the Appendix.

Since the size of the access overlap determines the relative weight of a data word written at one address when we compute the sums during a read operation at another address, it may be thought of as an intrinsic measure of the similarity or dissimilarity of two images used as addresses. Thus, the effective similarity of two images depends on the nature of the system, in particular the measure of distance in M, the choice of memory locations, and the chosen values of R. For example, if the set U_1 for a particular upper case "E" consists approximately of four disjoint spheres, each of volume p/4 (relative to M^+), and the set U_2 for a slightly different "E" consists approximately of four like spheres whose centers are near the centers of the spheres for the first "E", then the corresponding spheres will have large intersections.

This intrinsic measure of similarity also applies to images with different numbers of pieces, for example an "E" and an "F". Let U_1 be the set defined above for the "E", and let U_2 be the set for the "F", consisting approximately of three spheres, each of relative volume p/3. If we assume that in these particular instances of "E" and "F", the three pieces of the "F" exactly match three of the pieces of the "E", then each of the spheres in U_2 is concentric with one of the spheres in U_1. Since the radii of the spheres in U_2 are greater than those in U_1 (so that both

sets have the same volume), it follows that the intersection of U_1 and U_2 consists approximately of three of the spheres in U_1, each having a relative volume of p/4. Therefore, q, the relative volume of the intersection, is 3p/4, and the probability that three points chosen at random will lie in the intersection is $27p^3/64$, or about $0.42p^3$. The result of this calculation is that approximately 42% of the memory locations activated by the "E" are also activated by the "F" (and vice versa). However, since the spheres for an image are not completely disjoint, the program described below overestimates the volumes of these intersections, probably by about 10-20%. But since it does so for all of the volumes computed, the effect of this error is partially cancelled out.

A SMALL-SCALE SIMULATION

I have written a computer program to perform a small-scale approximate simulation of the Model 2 implementation. The program, called CREAD3, is written in BASIC, and runs on an IBM PC. This simulation does not have any hard memory locations as such. Instead, it simulates a read operation by computing the approximate size of the access overlap for the read address and each write address, and by then computing a weighted sum of the data assumed to be stored in the memory, as explained below. Since it is based on the assumptions and approximations described above, it is not highly accurate. However, it is easy to implement on a small computer, so I can carry out a variety of

simple experiments, and make changes in the design as I go along. The simulation can be used to test the performance of the memory design under various conditions. Thus it can provide some valuable insights, both into the nature of the encoding scheme and into a number of general issues concerning an SDM.

The program sets up a *current memory*, a working area that can hold two encoded images at a time. An image, represented as a set of segments and arcs, can be entered into current memory through the keyboard (Jaeckel, 1989a, p. 17). The program can also store an image in current memory onto a disk file, and can load an image into current memory from the disk file. When an image is entered into current memory, whether through the keyboard or from the disk file, it is centered and scaled (Jaeckel, 1989a, p. 17-18), and the five parameters used to represent each piece as a point in M are computed. Each piece is also converted to a 30-bit string, as described earlier.

To compare the two images in current memory, the program computes the Euclidean distance, as defined above, between each point in M representing a piece of the first image and each point representing a piece of the second image. It then finds the radii of the spheres about these points, as described above. The program then uses the formulas derived in the Appendix to compute the volume of the intersection of each sphere about a point for the first image with each sphere about a point for the second image. Under the simplifying assumptions above, the volume of the intersection of the two unions of spheres is approximately the sum of those computed volumes, and the size of the access

overlap for the two images is approximately proportional to the cube of that sum.

The memory is assumed to be trained by writing the characters in a training set to the memory. For each of these characters, its 256-bit encoding, described above, is assumed to be added to the counters for the memory locations activated by the character. The memory is then used to recognize an image of a character by reading from the memory using that image as the read address. In the simulation, the memory is not actually written to. Instead, a read operation is simulated by computing the approximate result of reading from the memory, as if it had been trained as described above.

As stated earlier, when we read from an SDM, we compute a vector of sums by adding the contents of the corresponding counters for the activated memory locations. These sums are made of multiple copies of the data words written to the memory, where the number of copies of each data word is equal to the number of memory locations activated by both the read address and the address at which that data word was written. Thus the resulting vector of sums is a weighted sum of the stored data words, each weighted by the size of the access overlap for the corresponding write address with the read address.

The program performs a simulated read operation by comparing each write address (representing a character assumed to be stored in the memory), one at a time, with the given read address, and computing the approximate size of the access overlap for that write address with the read address. It does this by computing

the volumes of the pairwise intersections of the spheres, as explained earlier. The cube of the sum of those volumes is approximately proportional to the expected number of memory locations activated by both images. The 256-bit data word for the character assumed to be written to the memory is multiplied by the size of the access overlap, and the corresponding products for all of the stored characters are added. The resulting vector of 256 sums is converted to bits as described below.

Since this simple simulation is based on some approximations, it is inexact because of the following sources of error: First, the spheres about the pieces of an image may not be disjoint, and they may extend beyond M^+; these factors cause the program to overestimate the volume of the intersection of the unions of spheres for two images, probably by about 10-20%. Second, the cube of the volume of the intersection is proportional to the *expected* number of hard memory locations in the access overlap — the actual number would differ somewhat because the hard memory locations would be chosen at random. Finally, if the counters for the memory locations have ceilings, and if in some counters the ceiling is reached while the data is being stored in the memory, then the contents of those counters will not be exactly the sum of the data written to those memory locations.

CONVERTING THE SUMS COMPUTED IN A READ OPERATION TO BIT STRINGS REPRESENTING SEGMENTS AND ARCS

If we store data using the 256-bit representation described earlier, then the result of a read operation is a vector of 256 sums, found by summing the contents of the corresponding counters for the activated memory locations. To interpret this vector as a representation of a stored character, we must convert it to a vector of bits, such that the patterns of 0's and 1's in each of the 30-bit blocks are of the form created by the method used to encode the pieces, and so that the other 16 sums indicate the starting and final blocks for the character found.

When a character to be written to the memory is encoded as a 256-bit string, each of the eight 30-bit blocks in the string may contain a piece of the character. Some of these eight blocks will contain pieces, and others may be blank (filled with 0's). Each block is divided into four sub-blocks, consisting of three six-bit strings, each containing a parameter value, and a 12-bit string, containing two parameter values. These five parameter values are each encoded as either two or three consecutive 1's within a sub-block, as explained in Jaeckel (1989a), p.34-39.

To convert the 256 sums to bits, the program does the following: First, for each sub-block in each 30-bit block, it computes all possible sums of two consecutive sums and sums of three consecutive sums. In the 12-component sub-blocks, since we think of the components as arranged in a circle, the sets of two or three consecutive sums are defined accordingly. In each of

the first three sub-blocks of each block, the object is to find the largest set of two or three consecutive sums. In the 12-component sub-blocks, we need to find the two largest non-overlapping sets of two or three consecutive sums, subject to the condition that the bit positions for these two sets represent a pair of angles at least 90^0 apart. (This condition is satisfied by any 12-bit string representing the shape and orientation of a piece.) To make the sums of two sums and the sums of three sums comparable, the program divides the sums of three sums by 1.333, a somewhat arbitrary number chosen to allow for some background noise. It then finds the largest sum of sums (or, for the pair of angles, the largest pair of such sums subject to the condition above) in each sub-block, and converts the components of these sums to 1's. The other components of each sub-block are converted to 0's. (Note that some of these computations could be done in parallel.)

Assume for a moment that the read image is similar to one and only one of the stored characters, which I will call the "target character". Each block of 30 bits now represents the system's best guess as to the parameters of the piece stored in that block, assuming that a piece of the target character is stored there. If a block is supposed to be blank, that is, if it does not contain a piece of the target character, then the random noise due to the other stored characters will cause the system to find an apparent piece there anyway. However, if no other stored character is very close to the read image, then the sums in those blocks should be small compared to the sums in the blocks

containing true pieces.

The next step is to decide which blocks actually contain pieces of the target character, and which should be considered blank. The program finds the largest sum among the sums for the eight starting-block bits, and calls the corresponding block the starting block. It then does the same for the final block. The program tentatively decides that the blocks containing the pieces to be found are those from the starting block to the final block, inclusive — wrapping around from the eighth block to the first if necessary. To confirm this decision, the program also computes a number for the *strength* of the piece found in each block. The strength of a piece found in a block is taken to be the minimum of the five sums of sums chosen as representing the parameter values, that is, the weakest of the five. If the strengths of the pieces in all of the blocks identified above as containing pieces are greater than the strengths of the pieces in all of the other blocks, then the decision as to which blocks contain the pieces is confirmed, and the pieces found in those blocks are taken to be the memory's response. If not, a warning message is displayed.

I used this somewhat rigid rule because of its simplicity and conservatism, and because I wanted to explore the general properties of an SDM without getting too deeply involved in the subtleties of the particular encoding scheme used here. More sophisticated ways of utilizing the information contained in the 256 sums could probably be devised.

RESULTS OF SIMULATION EXPERIMENTS

I did some experiments using the 20 characters shown in Figure 1 as the training set; that is, I assumed that the 256-bit encodings of these characters were written to the memory. Note that among these characters there are several groups of two or three similar characters. I then used the program to simulate read operations, as described above. That is, given an image that might be like one of the stored characters, the program would try to identify it.

As a first experiment, I tried reading from the memory with each of the characters in the training set — that is, using each stored character as a read address — to see if the memory could recognize it. I compared the results with the "right answer" by computing the Hamming distance between the pieces found, in the form of 30-bit strings, and the 30-bit encodings of the corresponding pieces of the stored character. (The Hamming distance is the number of bit errors in the pieces found.) At first I assumed that no "retraining" of the memory was done — that is, that each character was written to the memory once by adding its 256-bit encoding to the counters of the activated memory locations. For all of the characters, the blocks containing the pieces were correctly identified and confirmed. 15 of the 20 characters were recovered perfectly: The Hamming distance between the pieces found and the right answer was 0. For three of the characters the Hamming distance was 1; that is, there was one wrong bit among the pieces found. For the two

other characters, "P" and "R", the Hamming distances were 4 and 8 respectively, with most of the errors occurring in block 8. It is clear that these two similar characters, and perhaps some others, were interfering with each other, especially in that block. None of the pieces found in the other blocks were off by more than one bit.

Note that due to the method of encoding numbers as bit strings, an error of one bit in a block means that one of the five parameters will be off by only a small amount; so if the piece is reconstructed from the 30-bit string, it will not be off by much. This amount of error is tolerable at this stage of development. If each of the pieces found is close to the correct piece, the character in the training set could be approximately reconstructed from the results of the read. Each 30-bit string could be converted to the five parameter values describing a segment or arc by choosing, for each group of two or three consecutive 1's in the block, the midpoint of the interval represented by that group of bits. (See Jaeckel, 1989a, p. 35-38.) Using this information, the character could be approximately reproduced graphically.

I then added a small amount of retraining to the memory to improve its performance. A systematic method of retraining the memory would be to make several passes through the training set; during each pass we would measure the error in the memory's response for each item in the training set, and then make small changes in the contents of the counters, in a way that is intended to incrementally improve the memory's response. A

method of this kind was used by Joglekar (1989). Since there were serious errors with only two characters in the training set, I did not use a formal retraining method. Instead, I made small changes in the data assumed to be stored in some of the counters for block 8. When each character was originally written to the memory, each number added to a counter was 0 or 1. For the "P" I altered two of these numbers by ±0.15, and for the "R" I altered three of them by that amount. In other words, the contents of two of the counters for every memory location activated by "P", and of three of the counters for every memory location activated by "R", were altered by ±0.15, as if new write operations were done at these two addresses. These changes in the stored data are like the changes that would be made by a more systematic retraining method.

The rest of the experiments were done with the memory retrained in this way. I read from the memory again, using each character in the training set as the read address. This time the response for "P" was two bits off, one bit in each of two different pieces, and for "R" the response was four bits off, one bit in each of four pieces. For the other 18 characters the results were the same as before. Thus, after only a small amount of retraining, no individual piece of a character was in error by more than one bit. It seems likely that more retraining could remove most or all of the remaining errors. But the response now is good enough to permit some further experiments with the memory as it is.

I then read from the memory using as read addresses some

images of characters (shown in Figures 2 and 3) that are similar to characters in the training set. Some of them are distorted versions of characters in the training set, but with pieces that correspond in a one-to-one way to the pieces of those characters. Some are "noisy" — they have an extra piece, or are missing a piece. Two of the images in Figure 3 are in between two characters in the training set. I did not try distortions of the characters that would significantly change the way in which the image would be broken down into segments and arcs; the representation I used would not be expected to work in such cases.

I tried both single read operations and sequences of iterated read operations. A single read operation was sufficient to correctly recognize each of the characters in Figure 2. For each of them, the blocks containing the pieces of the right answer were correctly identified and confirmed, as described above, and for each such block the Hamming distance between the 30-bit string obtained as a result of the read operation and the 30-bit encoding of the corresponding piece of the right answer (that is, the number of wrong bits) was small. For five of the characters in Figure 2, there were no bit errors in any of their pieces. For the others, a few of the pieces found were one bit off, two pieces found for the "G" were two bits off, and one piece found for the "h" in Figure 2 was three bits off. None of the images in Figure 3 was recognized by doing a single read operation.

I then tried iterated read operations, both with those

characters in Figure 2 for which there were some errors in the pieces found, and with the characters in Figure 3. A sequence of iterated read operations with an autoassociative memory consists of using the result of each read — the response of the memory — as the read address for the next read. If the original read address is similar to a pattern (considered to be both an address and a data pattern) stored in the memory, the sequence of responses will sometimes converge to the stored pattern, as in Kanerva (1988), p. 68. Since an image can be approximately reconstructed from the 256-bit encoding used here, the memory system can be used like an autoassociative memory: We can do iterated read operations by converting the result of each read to an image and then using that image as the next read address.

The method of choosing the next read address when iterating is as follows: First the program decides which blocks lie between the starting block and the final block, as described above. If that decision is confirmed — that is, if those blocks are the ones with the greatest strengths — then the pieces found in those blocks are used. Otherwise, the program uses the pieces found in all of the blocks for which the strength is greater than or equal to the minimum strength for the blocks from the starting block to the final block. For each block used, the 30-bit string found in it is converted to the five parameter values describing a segment or arc. This is done by choosing, for each group of two or three consecutive 1's in the block, the midpoint of the interval represented by that group of bits. (See Jaeckel, 1989a, p. 35-38.) These segments and arcs may be thought of as

comprising an image; this image is used as the new read address.

The sequence of iterated reads was continued until the responses either converged to the right answer, or converged to something way off, or appeared not to be converging at all.

For five of the characters in Figure 2 there were some errors in the pieces found on the first read operation. In three cases, one iteration reduced the total Hamming distance (the number of wrong bits) to 0; in one case, the Hamming distance went from 1 to 2; and in one case — the "G" — the Hamming distance went from 5 to 11, and continuing to iterate caused the response to drift even farther away from the right answer. For the characters in Figure 3, the "m" and the "C" converged to the right answer in one or two iterations. The "  " converged to "A" after three iterations, rather than to "H". The "  " converged to "E" after one iteration, rather than to "S"; this is not surprising since three of its pieces are almost exactly like three of the pieces of the "E". Two of the characters converged to completely wrong characters after about six iterations, and two characters did not converge to anything.

These experiments can give us a feeling for the kinds of images that can be recognized by an SDM based on the method of representing images used here. Although more sophisticated iteration procedures might do better with some of the examples, I wanted to use a simple, automatic procedure in the initial experiments.

SOME DESIGN ALTERNATIVES

There are many possible variations and alternatives to the designs above. For example, there is the alternative to Model 2 mentioned earlier: To activate a memory location, we could require that each of its defining points be within R of a distinct piece of the image used as the address. (For images with fewer than three pieces, this rule would have to be modified.) This design should be better at distinguishing between images with some pieces in common, such as "E" and "F", because their access overlap, relative to the number of locations activated by a single image, would be smaller. For example, it can be shown that if the "F" is made of three of the four pieces comprising the "E", then their access overlap contains only 25% of the memory locations activated by the "E" (or by the "F"), instead of 42%, as was shown earlier for Model 2.

This alternative is more complex, however, in that it requires a little more work of the address decoders. When each address decoder computes the $3 \times NP$ matrix of distances between its defining points and the pieces of an image, it must look for a set of three small entries lying in distinct rows and in distinct columns. Also, the radius of the spheres would be a different function of the number of pieces in an image.

A read operation for this design may be simulated in a manner similar to the method above. The approximate size of the access overlap for two images may be estimated from the matrix of volumes (relative to M^+) of pairwise intersections of spheres as

follows: For each set of three entries in the matrix lying in distinct rows and in distinct columns, compute the product of those three volumes and multiply by six; this is the probability that three points chosen at random to be the defining points of a memory location will lie one in each of those three intersections. The sum of all of these products is approximately proportional to the expected number of memory locations in the access overlap. This computation would be subject to the same sources of error as the method used above for Model 2.

The advantage of this alternative design, a reduced access overlap for images that have some pieces in common, may be partially realized in Model 2 by choosing the three random defining points for a memory location so that they are unlikely to be very close to one another. This would increase the likelihood that the defining points of a memory location activated by an image would be near distinct pieces of the image. (However, in this case we could not do a simple simulation like the one above, because there would not be a simple way to estimate the size of the access overlap.)

If information about the "critical points" in an image is added to the representation, as described in Jaeckel (1989a), p. 48-52, then, in either of the SDM implementations above, the definition of the address space would have to be changed to correspond to this enhanced representation. Also, the rules for defining and activating the memory locations would have to be changed so that the set of memory locations activated by an image would depend on both the piece information and the critical point

information.

For the Model 1 implementation, we can choose a set of lattice points in C, the set of all possible critical points, say about 1000 of them, and add that many bit positions onto the 9072-bit vectors used to encode the pieces of an image. We must also define an appropriate distance function for points in C. Then, to encode the critical points of an image, we assign a 1 to every bit corresponding to a lattice point that is within some distance R of one or more of the points in C representing the critical points. The addresses and the activation rules for the memory locations would then be defined in terms of these very long binary vectors.

For the Model 2 implementation, we can define memory locations by choosing defining points from both M^+ and C (or an expanded version of C). For example, one or two defining points could be chosen from M^+, and two or one from C, for a total of three defining points for each memory location. The bit strings to be stored in the counters for the memory locations would be lengthened to create several blocks of bits for critical points, which would be encoded in a manner similar to that used for the pieces of the images. If we represent critical points by the method described in Jaeckel (1989a), p. 50-51, we could use six bits for each coordinate of the position of a critical point, and 24 bits for the set of directions in which segments or arcs radiate away from the point, for a total of 36 bits per critical point. As with the pieces, the encoded critical points can be stored in any of the blocks, as long as similar instances of a

character are stored in the same way.

The defining points for the memory locations in Model 2 could be chosen from a non-uniform probability distribution, perhaps based on the training data, as was suggested above for Model 1. Also, the memory locations in Model 2 could be based on different numbers of defining points. Another possibility is to activate a memory location if the sum, or some other function, of the distances from each defining point to the piece of the image nearest to it is less than some value, which would depend on the number of pieces. The pieces matched to the defining points might or might not be required to be distinct. Any of these design variations could be tried with the uniform metric, since the "spheres" would then be boxes, and it would be easy to compute the volumes of their intersections. There are other metrics that could also be tried.

More ambitious experiments could be done, using any of these methods. For example, the training set could contain several similar instances of each character, or pattern to be recognized, instead of one. In that case the result of a read operation might be a kind of average of the stored instances of the character. The capacity and the resolution of the memory under various conditions could be studied. Various methods of retraining the memory to improve its response could be tried. To obtain more accurate results, a more realistic simulation should be used. Finally, methods of representing and encoding broader classes of images — in particular, methods that capture more of the information in the images — need to be developed.

APPENDIX: THE VOLUME OF THE INTERSECTION OF TWO SPHERES

In order to estimate the size of the access overlap for Model 2, I needed to compute the volume of the intersection of two five-dimensional spheres. I will now derive a general formula for the volume of the intersection of two n-dimensional spheres.

By *n-dimensional sphere* I mean a sphere in n-dimensional Euclidean space, including its interior; the *volume* of such a sphere means the n-dimensional volume of its interior. The volume of an n-dimensional sphere of radius r is $V_n r^n$, where V_n is the volume of a sphere of unit radius. I will derive a formula for V_n below.

Suppose we have two n-dimensional spheres, the first with radius r and the second with radius s, and suppose the distance between their centers is d. This information determines the shape of their intersection. Note first that if $r + s \leq d$, the spheres are disjoint (or have one point in common), and if $d + s \leq r$ or $d + r \leq s$, one sphere lies entirely within the other. So we may assume that $|r - s| < d < r + s$, in which case the surfaces of the spheres intersect.

In order to have a clear picture, suppose that the center of the sphere of radius r is at the origin, that the center of the sphere of radius s is at the point d on the positive X_1-axis, and that $r \geq s$. Choose any point in the intersection of the *surfaces* of the two spheres. Let x_1 be its first coordinate, and let h be its distance from the X_1-axis. We

then have

$$x_1^2 + h^2 = r^2 \quad \text{and} \quad (d - x_1)^2 + h^2 = s^2 \ .$$

Solving for x_1, we find:

$$x_1 = \frac{d^2 + r^2 - s^2}{2d} \ .$$

Note that x_1 is the same for all such points. Therefore the intersection of the surfaces of the two spheres lies in a hyperplane orthogonal to the line of centers, whose distance from the center of the r-sphere is x_1. The distance of the hyperplane from the center of the s-sphere is

$$d - x_1 = \frac{d^2 - r^2 + s^2}{2d} \ .$$

The intersection of the spheres can now be divided at this hyperplane into two parts: a segment of the r-sphere to the right of the hyperplane, and a segment of the s-sphere to the left of it. So, to find the volume of the intersection, I will compute the volume of each segment separately, and then add them.

I will now find the volume of the part of the r-sphere to the right of the hyperplane. If for any x on the X_1-axis between x_1 and r, we slice through the sphere with a hyperplane orthogonal to the X_1-axis at x, the cross-section we find is an $(n-1)$-dimensional sphere with radius $\sqrt{r^2 - x^2}$. Its volume is V_{n-1} times the $(n-1)^{st}$ power of the radius. We can therefore integrate to find the volume of this segment of the r-sphere:

$$\text{Volume} = \int_{x_1}^{r} V_{n-1} \, (r^2 - x^2)^{(n-1)/2} \, dx \ .$$

If n is odd, we see that the integrand is a polynomial, so the integral can be evaluated easily, at least for small n. For the case $n = 5$, which is used in the program described above, I will transform the integral somewhat: Let $t = r - x$, the distance from the surface of the sphere to x. Then the volume of the segment becomes

$$V_4 \int_0^{t_1} [r^2 - (r - t)^2]^2 \, dt$$

$$= V_4 \int_0^{t_1} (4r^2t^2 - 4rt^3 + t^4) \, dt$$

$$= V_4 \, t_1^3 \cdot \left(\frac{4}{3}r^2 - rt_1 + \frac{1}{5}t_1^2\right) ,$$

where $t_1 = r - x_1$. A similar computation is done for the segment of the other sphere.

For the general case, if we let $x = r \cos \theta$, then $dx = -r \sin \theta \, d\theta$ and $r^2 - x^2 = r^2 \sin^2 \theta$, and the integral becomes

$$V_{n-1} \, r^n \int_0^{\theta_1} \sin^n \theta \, d\theta ,$$

where $\theta_1 = \cos^{-1}(x_1/r)$. This integral can be evaluated by using a standard recursion formula that reduces the power of the sine by two.

I will use this integral to derive a formula for V_n. Let $r = 1$, and let $\theta_1 = \frac{\pi}{2}$, which corresponds to $x_1 = 0$. Then the integral is the volume of half of an n-dimensional unit sphere:

$$\frac{1}{2} V_n = V_{n-1} \int_0^{\frac{\pi}{2}} \sin^n \theta \, d\theta \ .$$

Since this last integral may be found in a table of integrals, and is equal to

$$\frac{1}{2\sqrt{\pi}} \frac{\Gamma\left(\frac{n}{2} + \frac{1}{2}\right)}{\Gamma\left(\frac{n}{2} + 1\right)} \ ,$$

we have

$$V_n = V_{n-1} \cdot \sqrt{\pi} \frac{\Gamma\left(\frac{n}{2} + \frac{1}{2}\right)}{\Gamma\left(\frac{n}{2} + 1\right)} \ .$$

We can now show by induction that

$$V_n = \frac{\pi^{n/2}}{\Gamma\left(\frac{n}{2} + 1\right)} \ .$$

It is easy to see directly that $V_1 = 2$ and $V_2 = \pi$, and that these values agree with this formula. The following equation shows that if we assume that the formula is true for $n - 1$, then it is true for n:

$$V_n = \frac{\pi^{(n-1)/2}}{\Gamma\left(\frac{n-1}{2} + 1\right)} \cdot \sqrt{\pi} \frac{\Gamma\left(\frac{n}{2} + \frac{1}{2}\right)}{\Gamma\left(\frac{n}{2} + 1\right)} = \frac{\pi^{n/2}}{\Gamma\left(\frac{n}{2} + 1\right)} \ .$$

Therefore it is true for all n.

It follows that $V_3 = \frac{4}{3}\pi$, $V_4 = \frac{\pi^2}{2} = 4.934802$, and $V_5 = \frac{8\pi^2}{15}$ $= 5.263789$. These last two values are used by the program.

REFERENCES

Flynn, M. J., Kanerva, P., Ahanin, B., Bhadkamkar, N., Flaherty, P., & Hickey, P. (1988). Sparse Distributed Memory Prototype: Principles of Operation. Technical Report CSL-TR-87-338, Computer Systems Laboratory, Stanford University.

Hinton, G. (1981). Shape Representation in Parallel Systems. In *Proceedings of IJCAI 1981,* p. 1088-1096.

Jaeckel, L. A. (1989a). Some Methods of Encoding Simple Visual Images for Use with a Sparse Distributed Memory, with Applications to Character Recognition. RIACS Technical Report 89.29.

Jaeckel, L. A. (1989b). An Alternative Design for a Sparse Distributed Memory. RIACS Technical Report 89.28 (submitted for publication to *IEEE Transactions*).

Jaeckel, L. A. (1989c). A Class of Designs for a Sparse Distributed Memory. RIACS Technical Report 89.30.

Joglekar, U. (1989). Learning to Read Aloud: A Neural Network Approach Using Sparse Distributed Memory. RIACS Technical Report 89.27.

Kahan, S., T. Pavlidis, and H. S. Baird (1987). On the Recognition of Printed Characters of Any Font and Size. *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-9,* 274-288.

Kanerva, P. (1988). *Sparse Distributed Memory.* MIT Press, Cambridge, Mass.

Kanerva, P. (1989). A Cerebellar-Model Associative Memory as a Generalized Random-Access Memory. In *Proceedings of IEEE COMPCON 89.* IEEE Computer Society Press, Washington, D. C.

Keeler, J. D. (1988). Comparison Between Kanerva's SDM and Hopfield-type Neural Networks. *Cognitive Science, 12,* 299-329.

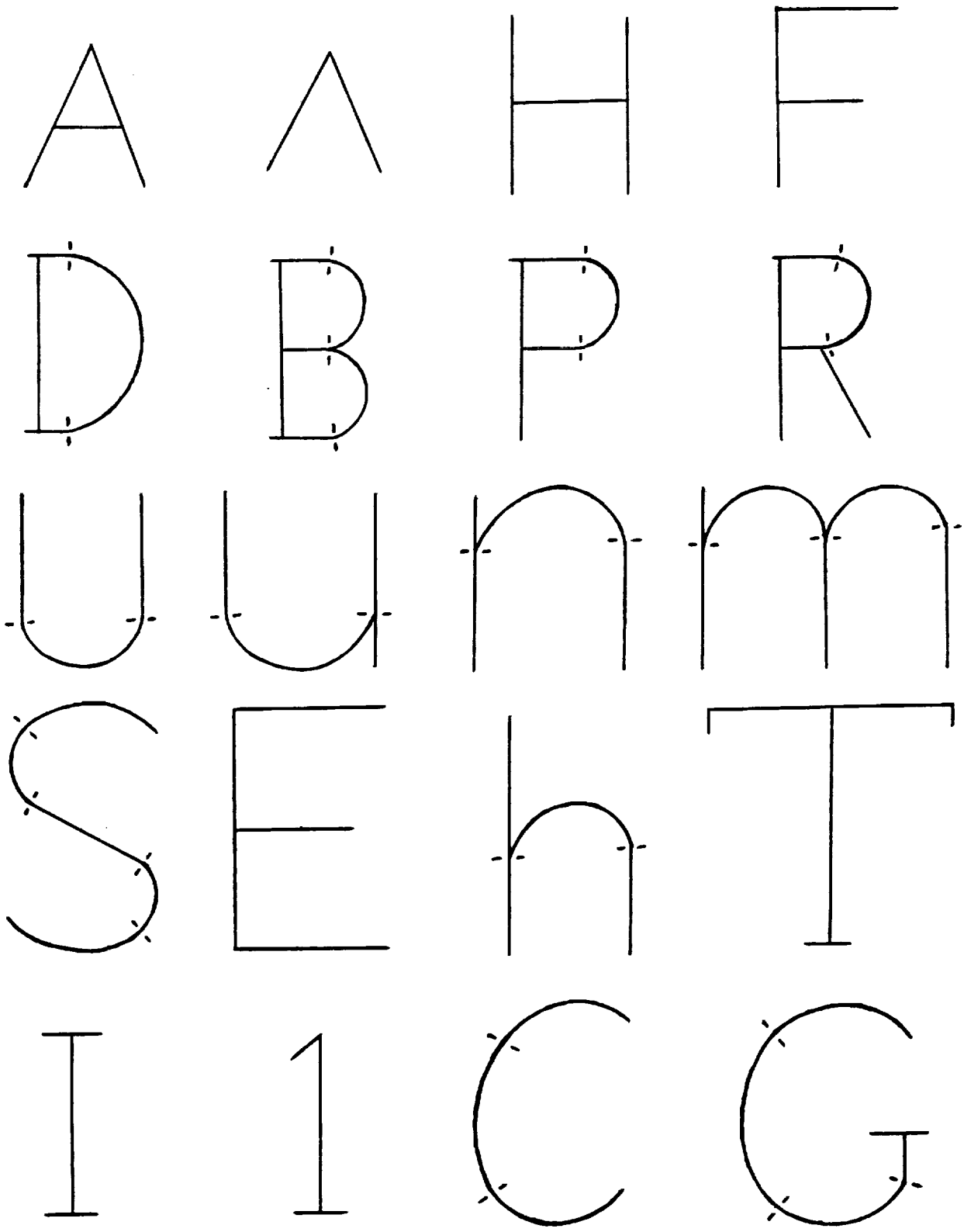

Figure 1: The training set.

The twenty characters assumed to be written to the memory. The tick marks (which do not appear in the actual images) show the points where the curves were divided into arcs.

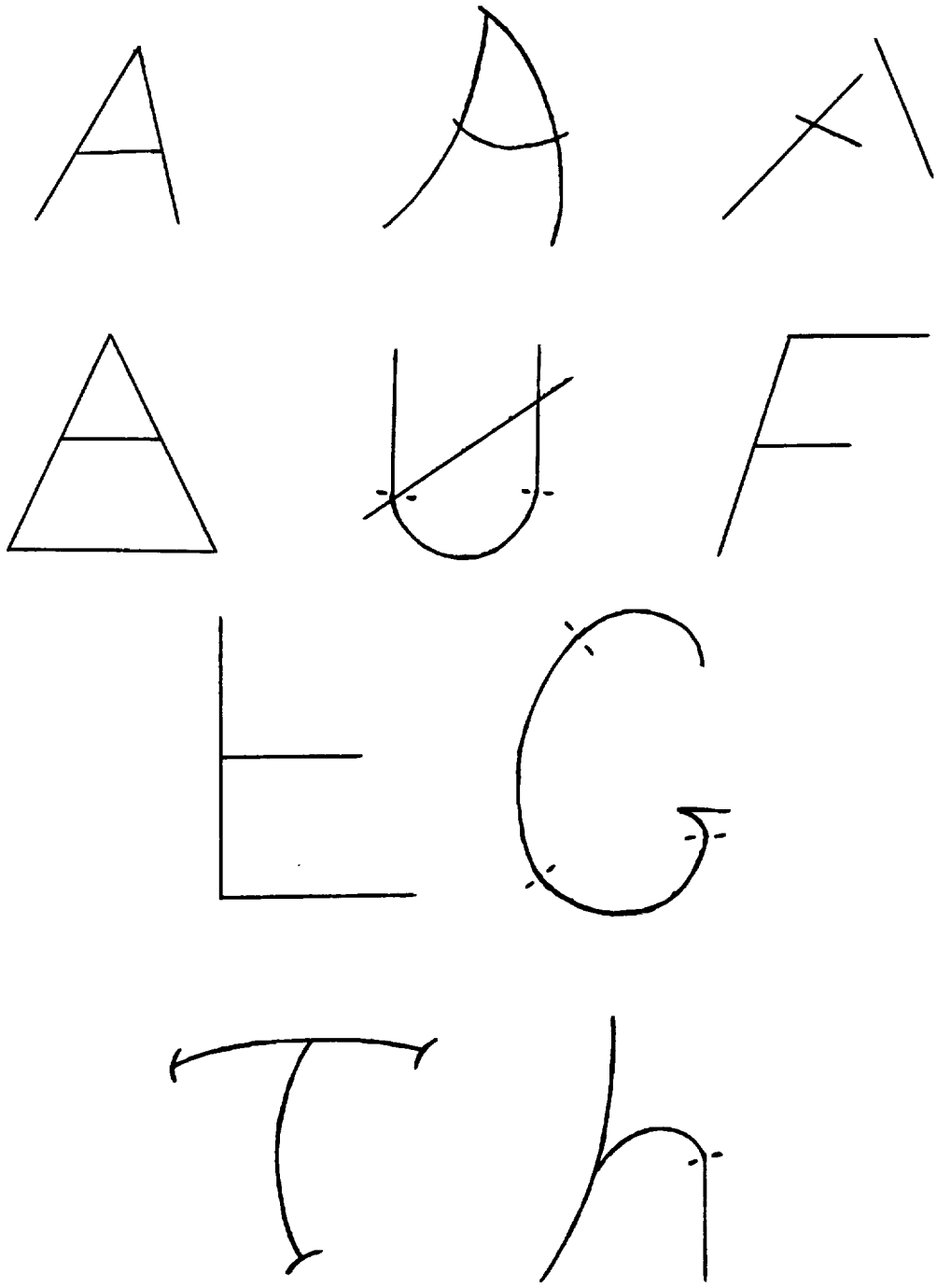

Figure 2. The system was able to correctly recognize each of these characters with one read operation. For a few of them, the response was improved by iterating.

Figure 3. The system could not recognize these characters with one read operation. A few of them were recognized after iterating.